Scalable Automated Proving and Debugging of Set-Based Specifications[*]

J.-F. Couchot[1] & D. Déharbe[2] & A. Giorgetti[1] & S. Ranise[3]

[1]LIFC, U. de Franche-Comté, Besançon (France)

[2]DIMAp/UFRN, Natal (Brazil)

[3]LORIA & INRIA-Lorraine, Nancy (France)

{couchot, giorgett}@lifc.univ-fcomte.fr, david@consiste.dimap.ufrn.br, ranise@loria.fr

Abstract

We present a technique to prove invariants of model-based specifications in a fragment of set theory. Proof obligations containing set theory constructs are translated to first-order logic with equality augmented with (an extension of) the theory of arrays with extensionality. The idea underlying the translation is that sets are represented by their characteristic function which, in turn, is encoded by an array of Booleans indexed on the elements of the set. A theorem proving procedure automating the verification of the proof obligations obtained by the translation is described. Furthermore, we discuss how a sub-formula can be extracted from a failed proof attempt and used by a model finder to build a counter-example. To be concrete, we use a B specification of a simple process scheduler on which we illustrate our technique.

Keywords: Set-theory, First-order logic with equality, Decision procedures, Superposition, BDDs, **haRVey**.

[*]Partially funded by INRIA/CASSIS project, CAPES grant BEX0006/02-5, and CNPq grant 500473/2003-0.

1 Introduction

Formal methods are increasingly integrated in the development cycle of both hardware and software artifacts. For software specification, industry is open to trying out rigorous notations like VDM [9], Z [18] or B [1]. Also, a combination of *theorem proving* and *model checking* is becoming increasingly popular to formally validate specifications.

Theorem proving discharges proof obligations entailing the correctness of a system with respect to its specification; it is a tedious activity requiring a significant amount of user interaction since it is usually conducted in undecidable logics. For example, Z and B are based on (variants of) set theory [1] which is well-known to be difficult to mechanise. State-of-the-art theorem provers (such as PVS[1]) provide only a limited amount of automation although a great deal of effort has been put into the automation of routine reasoning tasks. Indeed, a lot of research deals with the combination of decision procedures for selected theories and their incorporation in more general reasoning

[1]`http://pvs.csl.sri.com`

activities [17]. The main advantage of theorem proving is that it permits reasoning about infinite domains which are ubiquitous in software systems. The main disadvantage is that it can be difficult to say whether a property is not proved because the assumptions are not sufficiently strong or whether just some extra effort in theorem proving is required. *Model checking* consists of searching for a counter-example violating some property that the system is supposed to comply with. It can be made automatic for finite-state systems and only semi-automatic (i.e. the search may not terminate) for infinite-state systems. For infinite domains, the main drawback of model checking is that it can find counter-examples proving that the specification is contradictory with the system, but it may fail to prove that the specification is correct.

In this paper, we propose to leverage recent advances in the design of decision procedures for first-order theories [3, 10] to build automatic and flexible tools for proving and debugging set-based specifications. The key idea of our approach is that only fragments of set theory are used in many situations of practical relevance and such fragments can be translated into decidable theories of equational first-order logic. In order to test the feasibility of our approach, we have chosen the specification language of the B method. However, we intend the underlying method to be generally applicable to the model-based approach to specifications which encompasses also other notations such as Z or VDM.

The main ingredients of our method are fourfold. **First**, we translate a selected subset of the B specification language to first-order logic augmented with some set-theoretic constructs. More precisely, a B specification module—called Abstract Machine (AM)—is translated to first-order formulae encoding the relations between the before and after values of the variables of the AM according to its operations. Such formulae may contain sets (with a particular structure) and selected set-theoretic constructs. **Second**, the before-after representation of the system together with the invariant of the AM is translated into a set of first-order proof obligations (containing set-theoretic constructs) which entail that the invariant is inductive for the AM. (The first two ingredients of our method are briefly sketched in Section 2 since they are an adaptation of existing techniques, e.g. [1].) **Third**, we eliminate the set-theoretic constructs in the proof obligations by interpreting them in an extension of the decidable theory of arrays with extensionality (Section 3), see e.g. [3] . Such a translation is based on the idea that an array of Booleans indexed by the elements of a set s represents the characteristic function of s. **Fourth**, we pre-process the resulting proof obligations so as to eliminate quantifiers, thereby obtaining ground formulae (Section 4.1). Such pre-processing consists of exhaustively substituting a quantified sub-formula ψ with a propositional letter q and adding the axiom $q \Leftrightarrow \psi$ to the background theory. Afterwards, we invoke haRVey [10]—a reasoning system capable of proving the validity of quantifier-free formulae modulo equational first-order theories—to discharge the resulting proof obligations (Section 4.2). If a formula is shown to be valid, then we report it to the user. Otherwise, a selected sub-formula is extracted and passed to a model finder (i.e. a tool which takes a formula and attempts to find one of its models) so that a counter-example can be built and afterwards scrutinised by the user in order to understand why the formula failed to be proved valid (Section 4.3).

Related work. The closest related work is [15, 14] since it tries to combine the best of theorem proving and model finding by loosely coupling AtelierB[2] with the Alloy analyser[3]. The main difference is that the entire proof obligation is used for both theorem proving and model finding whereas we use theorem proving to simplify the formula so that only a small portion of it (ultimately responsible for its invalidity) is passed to a model finder, thereby considerably simplifying the task of the latter. There is some work (e.g. [4]) in using state-of-the-art theorem provers for formal reasoning in state-based specification languages such as B, Z, and VDM. The emphasis of such works is on the soundness of the translation from set theory to the logic used by the prover, ignoring the issues of automation thereby leaving the user with the burden of long and tedious interactive proofs. On the contrary, our work focuses on translating a fragment of set theory for which the theorem proving problem can be effectively automated by using decision procedures for first-order equational theories. To our knowledge, it is the first time that the idea of using (an extension of) a decision procedure for the theory of arrays is put forward to mechanise the reasoning in (fragments of) set theory by representing characteristic functions of sets with arrays. Section 4.4 reports some experiments which confirm the scalability of our approach on a class of large specifications manipulating simple data structures.

[2]`http://www.atelierb.societe.com/index.html`
[3]`http://sdg.lcs.mit.edu/alloy`

2 The B Specification and Verification Method

The B method has an associated specification notation called Abstract Machine Notation (AMN). This is a state-based notation similar to Z or VDM which features constructs such as assignments (`:=`), conditionals (`IF THEN ELSE`), multiple assignments (`||`), and non-deterministic choice (`ANY`) [1].

Roughly, an AM is composed of some state variables, an initialisation, and some operations that may alter the value of the state variables. Although the B method includes refinement and implementation of a specification, we consider here only the problem of checking whether an invariant is established by the initialisation and is preserved by the execution of all the operations of an AM. Proof obligations implying the correctness of the AM operations and initialisation with respect to its candidate invariant are generated following an effective procedure (along the lines in [1]).

Example 1 (The Process Scheduler) *As a running example on which we illustrate our techniques, we consider the process scheduler introduced in [11]. Although simple, this example allows us to discuss the typical problems arising in handling the type of AMs our technique is aimed at, i.e. (large) AMs which manipulate simple data structures, represented by sets of primitive elements. Its B specification is shown in Figure 1. At any one time, the system may have some processes* `ready` *to be scheduled, some processes* `waiting` *for some external action before they become* `ready` *and, possibly, a single* `active` *process. Each process is uniquely identified by an identifier (taken out of a set PID). The invariant states that* `active`*,* `ready`*, and*

```
MACHINE PSAM
SETS PID
VARIABLES active,ready,waiting
INVARIANT
  ready ⊆ PID ∧ active ⊆ PID ∧ waiting ⊆ PID ∧
  ready ∩ waiting = ∅ ∧ ready ∩ active = ∅ ∧
  active ∩ waiting = ∅ ∧ card(active) ≤ 1
INITIALISATION
  active,ready:= ∅,∅ || waiting:= PID
OPERATIONS
  Swap =
    IF active ≠ ∅ THEN waiting:= waiting ∪ active ||
      IF ready ≠ ∅ THEN
          ANY pr WHERE pr ∈ ready THEN
            active:= {pr} || ready:= ready - {pr}
          END
      ELSE active:= ∅
      END
   END
  Ready =
    ANY pw WHERE pw ∈ waiting
    THEN
      IF (active ≠ ∅) THEN
        ready:= ready ∪ {pw}
      ELSE
        active:= {pw}
      END
    END
END;
```

Figure 1: The Process Scheduler Abstract Machine.

`waiting` are pairwise disjoint subsets of `PID` and that at most one process can be in the `active` state (this enforces mutual exclusion in the execution of processes). The initialisation requires that all the processes are in the `waiting` state at the beginning. The operation `Swap` allows the AM to evolve by exchanging the currently `active` process with a `ready` one, leaving the system idle if there are no ready processes. In the operation `Ready`, a process pw*, that has been waiting, is somehow unblocked and can go to the ready state or to the active state, if the system is idle (we will see in Section 4.3 that the specification of this operation contains a bug).*

The specification of an operation such as `Swap` is a description of certain relevant properties that the intended state modification must fulfil. To formally express such properties, a common technique is to write the, so called, before-after predicate *which relates the values of the state variables (i.e. `ready`, `waiting`, and `active`) as they are immediately before and immediately after the operation takes place (see [1] for details). In order to write down such predicates, we adopt the convention of denoting the values of the variables just after the execution of the operation by priming the corresponding identifiers. The following before-after predicate* $P_{\mathtt{Swap}}(\mathtt{r}, \mathtt{w}, \mathsf{a}, \mathtt{r}', \mathtt{w}', \mathsf{a}')$ *specifies the operation `Swap`:*

$$\begin{array}{ll}
\textsf{if} & \mathsf{a} \neq \emptyset \\
\textsf{then} & \mathtt{w}' = \mathtt{w} \cup \mathsf{a} \wedge \\
& \quad \textsf{if} \;\; \mathtt{r} \neq \emptyset \;\; \textsf{then} \\
& \quad\quad \exists \mathtt{pr}.(\mathtt{pr} \in \mathtt{r} \wedge \mathsf{a}' = \{\mathtt{pr}\} \wedge \mathtt{r}' = \mathtt{r} - \{\mathtt{pr}\}) \\
& \quad \textsf{else} \\
& \quad\quad \mathsf{a}' = \emptyset \wedge \mathtt{r}' = \mathtt{r} \\
\textsf{else} & \mathsf{a}' = \mathsf{a} \wedge \mathtt{r}' = \mathtt{r} \wedge \mathtt{w}' = \mathtt{w}
\end{array} \tag{1}$$

where $\mathtt{r}, \mathtt{w}, \mathtt{a}, \mathtt{r}', \mathtt{w}', \mathtt{a}'$ *abbreviate* ***ready***, ***waiting***, ***active***, ***ready'***, ***waiting'*** *and* ***active'*** *respectively. The Boolean conditional connective* **if** A **then** B **else** C *abbreviates* $(A \Rightarrow B) \wedge (\neg A \Rightarrow C)$ *where* A, B, *and* C *are formulae. We use such a construct in order to preserve the structure of the B specification in the before-after predicate. Notice that the non-deterministic choice operator (ANY) is expressed by existential quantification and the multiple assignment operator (||) by conjunction. Also, state variables that are not explicitly assigned retain their previous value. Although this is not in the scope of this paper, we believe it would be easy to adapt our approach to handle other B constructs such as constants, machine parameters and properties.*

The invariant (cf. INVARIANT clause of Figure 1) can easily be translated to the following predicate:

$$\begin{aligned} & \mathtt{r} \subseteq \mathtt{p} \wedge \mathtt{w} \subseteq \mathtt{p} \wedge \mathtt{a} \subseteq \mathtt{p} \wedge \\ & \mathtt{r} \cap \mathtt{w} = \emptyset \wedge \mathtt{r} \cap \mathtt{a} = \emptyset \wedge \mathtt{a} \cap \mathtt{w} = \emptyset \wedge \\ & \mathtt{card}(\mathtt{a}) \leq 1, \end{aligned} \tag{2}$$

where $\mathtt{p}$ *is a constant representing PID; it is abbreviated by* $\mathsf{Inv}(\mathtt{r}, \mathtt{w}, \mathtt{a})$.

Once both an operation and the invariant have been specified by predicates, we can prove that the operation preserves the invariant by checking the validity of the following formula which encodes the fact that Inv *holds after the execution of* Swap, *provided that it holds before:*

$$\begin{aligned} \forall \mathtt{r}, \mathtt{w}, \mathtt{a}, \mathtt{r}', \mathtt{w}', \mathtt{a}'. \ & [\mathsf{Inv}(\mathtt{r}, \mathtt{w}, \mathtt{a}) \wedge P_{\mathtt{Swap}}(\mathtt{r}, \mathtt{w}, \mathtt{a}, \mathtt{r}', \mathtt{w}', \mathtt{a}') \\ & \Rightarrow \mathsf{Inv}(\mathtt{r}', \mathtt{w}', \mathtt{a}')]. \end{aligned} \tag{3}$$

In addition to proving that each operation preserves the invariant Inv, *one also has to check that* Inv *is satisfied by the initialisation condition (cf. INITIALISATION clause of Figure 1). We omit the corresponding proof obligation, since this does not add much to the discussion.*

Proof obligations—such as (3)— shall be discharged by using automated theorem provers. Typically, in currently available commercial tools supporting the B method, there is a number of proof obligations that the automated prover cannot discharge so the developer can switch the prover to an interactive mode and attempt to try to discharge the remaining proof obligations manually. The B method is actually supported by two commercially available tools: the B-Toolkit[4] and the AtelierB.[5] Although quite successful, both tools leave the developer without help to discover why a certain proof obligation has failed to be shown valid.

In the rest of this paper, we describe a technique to check the validity of proof obligations and to provide the user with counter-examples when the validity check fails.

3 Translating Set-Based Proof Obligations

In Section 2, we have relied on the reader's intuition of basic concepts of first-order logic and naive set theory to write down the before-after predicates specifying the operation, the invariant, and the proof obligation of the AM in Figure 1. Here, we define the simple version of set theory we use and explain how to translate it into a suitable extension of first-order logic with equality so that haRVey (cf. Section 4.2)—a system to check the validity of equational first-order

[4] `http://www.b-core.com/btoolkit.html`
[5] `http://www.atelierb.societe.com/index.html`

formulae—can be used to discharge such proof obligations (if possible).

Below, we assume the usual syntactic and semantic notions of first-order logic with equality as defined for example in [12]. We say that a formula ϕ is *satisfiable modulo a theory* $\mathcal{T}$ iff $\phi \wedge \mathcal{T}$ is satisfiable.[6]

3.1 A Simple Set Theory

For simplicity, in this paper, we consider a restricted fragment of set theory, which we denote with *SSET*. Notice, however, that our approach can be extended to handle more expressive fragments of set theory such as the theory of Hereditarily Finite Sets with Atoms (see, e.g. [5]), which permits sets of primitive elements, sets of sets of primitive elements, and so on. *SSET* is a theory of first-order sorted logic with equality. It contains two distinct sort symbols ELEM and SET. Its set of terms contains the variable and constant symbols of sort ELEM and SET. We assume there is at least one constant of sort ELEM. The distinguished constant $\emptyset$ is a term of sort SET. If e is a term of sort ELEM, then $\{e\}$ is a term of sort SET; also, if s_1 and s_2 are terms of sort SET, then $s_1 \bowtie s_2$ is also a term of sort SET, where $\bowtie$ is one of the binary function symbols $\cap$ (intersection), $\cup$ (union), and $\setminus$ (set difference). We also write $\{e_1, \ldots, e_n\}$ as an abbreviation of $\{e_1\} \cup \cdots \cup \{e_n\}$. The set of atoms of *SSET* contains expressions of the form $e_1 = e_2$, $e \in s$, $s_1 \subseteq s_2$, $s_1 = s_2$, where e, e_1, e_2 are terms of sort ELEM and s, s_1, s_2 are terms of sort SET. Literals, Boolean combinations of literals, and possibly quantified formulae are inductively defined in the usual way. Furthermore, let $Ax(SSET)$ be the set obtained by adding the following axioms to the theory of equality:

$$\forall E.(\neg E \in \emptyset), \quad (4)$$

$$\forall E.(E \in \{E\}), \quad (5)$$

$$\forall E, F.(E \neq F \Rightarrow \neg E \in \{F\}), \quad (6)$$

$$\forall E, S_1, S_2.(E \in S_1 \cup S_2 \Leftrightarrow (E \in S_1 \vee E \in S_2)), \quad (7)$$

$$\forall E, S_1, S_2.(E \in S_1 \cap S_2 \Leftrightarrow (E \in S_1 \wedge E \in S_2)), \quad (8)$$

$$\forall E, S_1, S_2.(E \in S_1 \setminus S_2 \Leftrightarrow (E \in S_1 \wedge \neg E \in S_2)), \quad (9)$$

$$\forall S_1, S_2.(S_1 \subseteq S_2 \Leftrightarrow \forall E.(E \in S_1 \Rightarrow E \in S_2)), \quad (10)$$

$$\forall S_1, S_2.(S_1 = S_2 \Leftrightarrow \forall E.(E \in S_1 \Leftrightarrow E \in S_2)), \quad (11)$$

where E, F are variables of sort ELEM and S_1, S_2 are variables of sort SET. The semantics of *SSET* is given by the class $\mathcal{I}(SSET)$ of first-order interpretations satisfying each axiom in $Ax(SSET)$ whose domain interpreting sort SET is "generated by" $\emptyset$, $\{\_\}$, and $\cup$; i.e. all elements of the domain of sort SET are the interpretations of terms containing only $\emptyset$, $\{\_\}$, and $\cup$ as function symbols whose application yields terms of sort SET.[7] This allows us to prove properties by induction over $\emptyset$, $\{\_\}$, and $\cup$ (see, e.g. Fact 1 below).

Intuitively, the set theory we are considering permits to reason about sets with a very simple structure, i.e. sets (represented by terms of sort SET) which are subsets of a given universal set (e.g. the set `PID` in the example of Figure 1) and whose elements (represented by terms of sort ELEM) are primitive. So, for example, if the universal set is that of integers, the set $\{1, 2, 3\}$ is a valid set of *SSET*, whereas $\{1, \{2\}\}$ is not. Notice that *SSET* is already useful in many practical verification problems involving large systems (say some pages of B specifications) which manipulate simple data structures represented by sets of primitive elements.

[6] A theory is a set of formulae closed under logical consequence.

[7] This is equivalent to say that we adopt an "initial model semantics" for *SSET*.

3.2 The Theory of Arrays with Extensionality

Let $\mathcal{A}_s^e$ be the many-sorted theory with sorts VALUE, INDEX and ARRAY, with function symbols write (abbreviated below with wr) and read (abbreviated below with rd) of type ARRAY $\times$ INDEX $\times$ VALUE $\longrightarrow$ ARRAY and ARRAY $\times$ INDEX $\longrightarrow$ VALUE respectively. Furthermore, let $Ax(\mathcal{A}_s^e)$ be the set of axioms obtained by adding the following axioms to the theory of equality:

$$\forall A, I, E.(\mathsf{rd}(\mathsf{wr}(A, I, E), I) = E), \quad (12)$$

$$\forall A, I, J, E.(I \neq J \Rightarrow \mathsf{rd}(\mathsf{wr}(A, I, E), J) = \mathsf{rd}(A, J)), \quad (13)$$

$$\forall A, B.(\forall I.(\mathsf{rd}(A, I) = \mathsf{rd}(B, I)) \Rightarrow A = B), \quad (14)$$

where A and B are variables of sort ARRAY, I and J are variables of sort INDEX, and E is a variable of sort VALUE. $\Sigma_{\mathcal{A}_s^e}$ denotes a signature containing the function symbols rd, wr, and a finite set of constant symbols. We assume that the signature of $\mathcal{A}_s^e$ admits at least one ground term for each sort. Checking the satisfiability of conjunctions of ground literals modulo $\mathcal{A}_s^e$ is decidable (see, e.g. [3]).

3.3 From Set Theory to Array Theory

We explain how to translate formulae of *SSET* to (extensions of) $\mathcal{A}_s^e$ so that the reasoning system haRVey (cf. Section 4.2) can be used to discharge the proof obligations which imply that an invariant of an AM is an inductive property of the machine (along the lines sketched in Section 2).

The intuition underlying our approach is based on using the characteristic function to represent sets. Such a function, in turn, can be encoded by an array of Booleans whose indexes are the elements of the set. For example, the set $s := \{1, 2\}$ can be represented as $s[1] = s[2] = \mathit{true}$ and $s[x] = \mathit{false}$, for all x distinct from 1 and 2.

First, let $\mathcal{BA}_s^e \supset \mathcal{A}_s^e$ be an extension of $\mathcal{A}_s^e$ containing two distinguished constants tt and ff of sort VALUE together with the axiom

$$\mathsf{tt} \neq \mathsf{ff}. \quad (15)$$

In this way, we consider arrays storing Boolean values (from this, the '$\mathcal{B}$' in front of '$\mathcal{A}_s^e$' in '$\mathcal{BA}_s^e$'). Furthermore, $\mathcal{BA}_s^e$ contains a distinguished constant symbol mty of sort ARRAY and the axiom

$$\forall I.(\mathsf{rd}(\mathsf{mty}, I) = \mathsf{ff}), \quad (16)$$

where I is a variable of sort INDEX (the intuition is that mty is the counterpart of the empty set in $\mathcal{A}_s^e$).

Then, we define three functions S, T, and F from the sorts, terms, and formulae of *SSET* to the sorts, pairs of terms and set of formulae, and pairs of formulae and set of formulae (respectively) of $\mathcal{BA}_s^e$. Below, we will see that other symbols and axioms will be added to $\mathcal{BA}_s^e$ in the process of translating a formula of *SSET* to a formula in first-order logic with equality by applying functions T and F. Formally, this is done by returning the translated terms together with the set of formulae to be added to $\mathcal{BA}_s^e$. Such formulae will implicitely determine the symbols to be added to $\mathcal{BA}_s^e$.

We define S(ELEM) := INDEX and S(SET) := ARRAY. Then, we assume that T is s.t. it maps (one-to-one) constants of sort ELEM and SET to constants of sort INDEX and ARRAY, respectively. Now, we homeomorphically extend T to compound terms of *SSET* as follows, where i and u are fresh constants of sort INDEX and ARRAY, respectively:

$\mathsf{T}(\emptyset) := (\mathsf{mty}, \emptyset)$;

$\mathsf{T}(x) := (i, \emptyset)$ if x is of sort ELEM;

$\mathsf{T}(x) := (u, \emptyset)$ if x is of sort SET;

$\mathsf{T}(\{e\}) := (u, \{u = \mathsf{wr}(\mathsf{mty}, e, \mathsf{tt})\})$,

$\mathsf{T}(e_1 \cup e_2) := (u, \{a_\cup\} \cup \alpha_1 \cup \alpha_2)$,

$\mathsf{T}(e_1 \cap e_2) := (u, \{a_\cap\} \cup \alpha_1 \cup \alpha_2)$, and

$\mathsf{T}(e_1 \setminus e_2) := (u, \{a_\setminus\} \cup \alpha_1 \cup \alpha_2)$, where

$a_\cup$ is $\forall I.(\mathsf{rd}(\widehat{e_1}, I) = \mathsf{tt} \vee \mathsf{rd}(\widehat{e_2}, I) = \mathsf{tt} \Leftrightarrow \mathsf{rd}(u, I) = \mathsf{tt})$, $a_\cap$ is $\forall I.(\mathsf{rd}(\widehat{e_1}, I) = \mathsf{tt} \wedge \mathsf{rd}(\widehat{e_2}, I) = \mathsf{tt} \Leftrightarrow \mathsf{rd}(u, I) = \mathsf{tt})$, $a_\setminus$ is $\forall I.(\mathsf{rd}(\widehat{e_1}, I) = \mathsf{tt} \wedge \mathsf{rd}(\widehat{e_2}, I) = \mathsf{ff} \Leftrightarrow \mathsf{rd}(u, I) = \mathsf{tt})$ and $\mathsf{T}(e_j) = (\widehat{e_j}, \alpha_j)$ for $j = 1, 2$. We regard $\{e_1, ..., e_n\}$ (for $n > 1$) as an abbreviation of $\{e_1\} \cup \cdots \cup \{e_n\}$.

Then, for each constant u of sort ARRAY (representing a set), we add to $\mathcal{BA}^e_s$ the following axiom:

$$\forall I.(\mathsf{rd}(u, I) = \mathsf{tt} \vee \mathsf{rd}(u, I) = \mathsf{ff}), \qquad (17)$$

where I is a variable of sort INDEX. Axiom (17) constrains the co-domain of the characteristic function represented by u to be $\{\mathsf{tt}, \mathsf{ff}\}$.

Furthermore, we translate the ground atoms of *SSET* by defining the function F as follows:

$\mathsf{F}(e_1 \in e_2) := (\mathsf{rd}(\widehat{e_2}, \widehat{e_1}) = \mathsf{tt}, \alpha_1 \cup \alpha_2)$,

$\mathsf{F}(e_1 = e_2) := (\widehat{e_1} = \widehat{e_2}, \alpha_1 \cup \alpha_2)$,

$\mathsf{F}(e_1 \neq e_2) := \mathsf{F}(\neg(e_1 = e_2))$,

$\mathsf{F}(e_1 \subset e_2) := \mathsf{F}(e_1 \subseteq e_2 \wedge e_1 \neq e_2)$,

$\mathsf{F}(e_1 \subseteq e_2) := (q, \alpha_1 \cup \alpha_2 \cup \alpha)$,

where q is a fresh propositional letter, $\mathsf{T}(e_i) = (\widehat{e_i}, \alpha_i)$ for $i = 1, 2$ and α is $\{q \Leftrightarrow \forall X.(\mathsf{rd}(\widehat{e_1}, X) = \mathsf{tt} \Rightarrow \mathsf{rd}(\widehat{e_2}, X) = \mathsf{tt})\}$.

Then, the translation process is homeomorphically extended to Boolean combinations of atoms in the following way:

$\mathsf{F}(\neg\phi) := (\neg\,\widehat{\phi}, \alpha)$ where $\mathsf{F}(\phi) = (\widehat{\phi}, \alpha)$,

$\mathsf{F}(\phi_1 \diamond \phi_2) := (\widehat{\phi_1} \diamond\ \widehat{\phi_2}, \alpha_1 \cup \alpha_2)$

where $\mathsf{F}(\phi_i) = (\widehat{\phi}_i, \alpha_i)$ for $i = 1, 2$ and $\diamond$ stands for either $\vee, \wedge, \Rightarrow$ or $\Leftrightarrow$, and

$\mathsf{F}(\mathsf{if}\ \phi_1\ \mathsf{then}\ \phi_2\ \mathsf{else}\ \phi_3) := (\mathsf{if}\ \widehat{\phi}_1\ \mathsf{then}\ \widehat{\phi}_2\ \mathsf{else}\ \widehat{\phi}_3, \alpha_1 \cup \alpha_2 \cup \alpha_3)$

where $\mathsf{F}(\phi_i) = (\widehat{\phi}_i, \alpha_i)$ for $i = 1, 2$, and 3.

Now, we are in the position to state the main property of our translation process (we state it in terms of satisfiability since our theorem proving approach is based on refutation; see Section 4.2). Let ϕ be a ground formula of *SSET* and $(\widehat{\phi}, \alpha) = \mathsf{F}(\phi)$ be its translation. We denote by $\mathcal{BA}^e_s$ the theory containing $\mathcal{A}^e_s$, the axioms (15), (16), (17) and the formulae in α.

Theorem 1 *ϕ is satisfiable modulo SSET iff $\widehat{\phi}$ is satisfiable modulo $\mathcal{BA}^e_s$.*

In order to prove the Theorem, we need some technical definitions and facts.

Fact 1 *For any ground term t of SSET, there exists a term t' containing only the function symbols $\emptyset$, $\{\_\}$, and $\cup$ s.t. $t = t'$ is satisfied by an interpretation in $\mathcal{I}(SSET)$.*

The proof consists of an easy induction on $\emptyset$, $\{\_\}$, and $\cup$. Then, we introduce the function ins as follows:

$$\mathsf{ins}(e, \emptyset) = \{e\} \qquad (18)$$

$$\mathsf{ins}(e, s) = \{e\} \cup s, \qquad (19)$$

for all ground terms e and s of sort ELEM and SET, respectively.

Fact 2 *Let $\mathcal{R}$ be the term rewriting system consisting of (18) and (19), oriented from right to left, and* $\mathsf{ins}(e, \mathsf{mty}) \cup s = \mathsf{ins}(e, s)$*, oriented from left to right. Then, $\mathcal{R}$ is terminating and confluent.*

The termination of $\mathcal{R}$ can be easily seen by observing that the number of occurrences of $\{\_\}$ and $\cup$ decreases following the orientation of each rewrite rule. It is also trivial to see that if a term t does not contain any occurrence of $\{\_\}$ or $\cup$, then no rule in $\mathcal{R}$ can be applied. The fact of being confluent is an immediate consequence of the fact that the set of equations is closed under the rules of the superposition calculus [16].[8] As a consequence, we are entitled to define a function $\cdot'$ on terms of $SSET$ obtained by using Facts 1 and 2. By abuse of notation, we extend such a function to formulae as follows: ϕ' will denote the formula obtained from ϕ by applying $\cdot'$ to its sub-terms. Then, we introduce the set $Ax(SSET')$ of axioms obtained from $Ax(SSET)$ by replacing (5) and (6) with

$$\forall E, S.(E \in \mathsf{ins}(E, S)), \quad (20)$$

$$\forall E, F, S.(E \neq F \Rightarrow (E \in \mathsf{ins}(F, S) \Leftrightarrow E \in S)), \quad (21)$$

where E, F are variables of sort ELEM and S is a variable of sort SET. The signature of $SSET'$ is obtained from that of $SSET$ by replacing $\{\_\}$ with ins.

Fact 3 *ϕ is $SSET$-satisfiable iff ϕ' is $SSET'$-satisfiable.*

To see this, it is sufficient to observe that both (20) and (21) are logical consequences of $Ax(SSET)$ and of (18) and (19). Now, we consider the mapping T' from ground terms of $SSET'$ to pairs of ground terms and set of formulae of $\mathcal{BA}_s^e$ defined as follows, where i and u are fresh constants of sort INDEX and ARRAY, respectively:

$\mathsf{T}'(\mathsf{ins}(e, s)) := (\mathsf{wr}(\widehat{s}, e, \mathsf{tt}), \alpha)$ where $\mathsf{T}'(s) = (\widehat{s}, \alpha)$;

$\mathsf{T}'(\emptyset) := (\mathsf{mty}, \emptyset)$;

$\mathsf{T}'(x) := (i, \emptyset)$ if x is of sort ELEM;

$\mathsf{T}'(x) := (u, a_u)$ if x is of sort SET, where a_u is $\forall I.(\mathsf{rd}(u, I) = \mathsf{tt} \vee \mathsf{rd}(u, I) = \mathsf{ff})$, and I is a variable of sort INDEX;

$\mathsf{T}'(e_1 \cup e_2) := (u, \{a_\cup\} \cup \alpha_1 \cup \alpha_2)$, where $a_\cup$ is $\forall I.(\mathsf{rd}(\widehat{e_1}, I) = \mathsf{tt} \vee \mathsf{rd}(\widehat{e_2}, I) = \mathsf{tt} \Leftrightarrow \mathsf{rd}(u, I) = \mathsf{tt})$ and $\mathsf{T}'(e_j) = (\widehat{e_j}, \alpha_j)$ for $j = 1, 2$.

Lemma 1 *Let t be a ground term on the signature of $SSET$ and t' be the translated term on the signature of $SSET'$. Let $\mathsf{T}(t) = (s, \alpha)$ and $\mathsf{T}'(t') = (s', \alpha')$. Then, $s = s'$ is a logical consequence of $\mathcal{A}_s^e$, α, α', (15), (16) and (17).*

Again, the proof is obtained by induction on the structure of the term t and therefore is omitted. If we define F' as F above where the calls to T are replaced with calls to T', then we can easily derive that ψ is logically equivalent to ψ' modulo $\mathcal{A}_s^e \cup \{\alpha, \alpha'\}$ as a corollary of Lemma 1, where $(\psi, \alpha) = \mathsf{F}(\phi)$ and $(\psi', \alpha') = \mathsf{F}(\phi')$. The following diagram summarizes the situation:

$$\begin{tikzcd} SSET \arrow[r, ".'"] \arrow[dr, "\mathbf{F}"'] & SSET' \arrow[d, "\mathbf{F}'"] \\ & \mathcal{BA}_s^e \end{tikzcd}$$

Proof of Theorem 1. We prove (the counterpositive of) if ϕ is satisfiable modulo $SSET$ then $\widehat{\phi}$ is satisfiable modulo $\mathcal{BA}_s^e$. First of all, we translate the formula ϕ of $SSET$ to a formula ϕ' of $SSET'$ by using Fact 1 and Fact 2. Then, we observe that the translation of the ground instances of the axioms in $Ax(SSET')$ (under F') are all logical consequences of $Ax(\mathcal{BA}_s^e)$. If $\widehat{\phi'}$ is the formula of $\mathcal{BA}_s^e$ which is the translation of ϕ' under F', then, by Fact 3, Lemma 1, and the previous observation, we are entitled to conclude that ϕ is unsatisfiable modulo $SSET$ if $\widehat{\phi}$ is unsatisfiable modulo $\mathcal{BA}_s^e$. The other implication of the biconditional is similar and therefore omitted. □

[8] In fact, $\mathsf{ins}(e, \mathsf{mty}) \cup s = \mathsf{ins}(e, s)$ is obtained by superposition of (18) and (19).

3.4 Two Important Extensions

In order to enlarge the scope of applicability of our technique, we consider two extensions of the translation defined above. The former handles (a restricted form of) the cardinality operator which is frequently used in set-based specification. The latter considers non-ground formulae containing quantifiers over the elements of the universal set. We do not consider quantifiers over sets since it is well-known that the full first-order theory of arrays is undecidable (see [3] for further references on this point).

The cardinality operator. Let us consider only ground atoms of the form $\mathsf{card}(s) = k$, where s is a term of sort SET and k is a given numeral. Then, we can replace each atom of the form $\mathsf{card}(s) = k$ with $s = \{f_1, ..., f_k\}$, where f_i is a fresh constant of sort ELEM (for $i = 1, ..., k$). After the exhaustive application of such a rule we obtain a formula of *SSET* which can be translated to $\mathcal{BA}_s^e$ as described above. We can generalise the approach to handle more complex arithmetic relation. For example, $\mathtt{card(active)} \leq \mathtt{1}$ can be rewritten as $\mathtt{card(active)} = \mathtt{0} \vee \mathtt{card(active)} = \mathtt{1}$ which, in turn, rewrites to $\mathtt{active} = \emptyset \vee \mathtt{active} = \{\mathtt{f_1}\}$, where $\mathtt{f_1}$ is a fresh constant of sort ELEM. Formally, we extend the definition of F above as follows:

$$\mathsf{F}(\mathtt{card}(e) = 0) := (\widehat{e} = \mathsf{mty}, \alpha)$$

where $\mathsf{T}(e) = (\widehat{e}, \alpha)$,

$$\mathsf{F}(\mathtt{card}(e) = k) := (q, \alpha \cup \{q \Leftrightarrow \exists x_1, \cdots, x_k.\beta\})$$
$$\mathsf{F}(\mathtt{card}(e) \leq k) := (q, \alpha \cup \{q \Leftrightarrow \exists x_1, \cdots, x_k.\varphi\})$$

where q is a fresh propositional letter, k is a numeral constant denoting an integer s.t. $k \neq 0$, $\mathsf{T}(e) = (\widehat{e}, \alpha)$, φ is $\widehat{e} = \mathsf{wr}(\cdots \mathsf{wr}(\mathsf{mty}, x_1, \mathsf{tt}) \cdots, x_k, \mathsf{tt})$, and β is $\bigwedge_{1 \leq i < j \leq k} (x_i \neq x_j) \wedge \varphi$.

By applying the above three equations defining the extension of F, we can eliminate all occurrences of the cardinality operator in a formula. As a consequence, we obtain a formula of *SSET* to which Theorem 1 is still applicable. In other words, Theorem 1 can be straightforwardly adapted to hold for the extension of *SSET* with the restricted form of the cardinality operator considered here.

Quantifiers over set elements. Frequently, the proof obligations resulting from the verification of invariants of B machines are not ground. They usually contain quantifiers over the elements of a certain set. For example, we will obtain existentially quantified sub-formulae whenever we handle the non-deterministic choice operator (i.e. `ANY`), see the Example below. To handle quantifiers over elements of sets, we define the function F^Q to be equal to F on ground formulae and s.t.

$$\mathsf{F}^Q(Qx.\phi) := (Qx.\widehat{\phi}, \alpha)$$

where x is a variable of sort ELEM, Q is either $\forall$ or $\exists$, and $\mathsf{F}^Q(\phi) = (\widehat{\phi}, \alpha)$.

Example 2 *(*The Process Scheduler—continued*) The before-after predicate $P_{\mathtt{Swap}}$ in (1) is translated to* *the predicate* $P^{eq}_{\mathtt{Swap}}$*:*

```
if  $\mathsf{a} \neq \mathsf{mty}$  then
  $\mathtt{w}' = \mathtt{w\_U\_a} \wedge$
  if  $\mathtt{r} \neq \mathsf{mty}$  then
    $\exists \mathtt{pr}.(\mathsf{rd}(\mathtt{r}, \mathtt{pr}) = \mathsf{tt} \wedge \mathsf{a}' = \mathsf{wr}(\mathsf{mty}, \mathtt{pr}, \mathsf{tt}) \wedge$
        $\mathtt{r}' = \mathsf{wr}(\mathtt{r}, \mathtt{pr}, \mathsf{ff}))$
  else
    $\mathsf{a}' = \mathsf{mty} \wedge \mathtt{r}' = \mathtt{r}$
else
  $\mathsf{a}' = \mathsf{a} \wedge \mathtt{r}' = \mathtt{r} \wedge \mathtt{w}' = \mathtt{w}$
```

(22)

where $\mathtt{w}, \mathsf{a}, \mathtt{r}$ *and their primed versions are variables of sort* ARRAY*.*

It is possible to express the set-theoretic constructs in the verification condition (3) by using the translation described above, thereby obtaining the formula

$$\forall \mathtt{r}, \mathtt{w}, \mathsf{a}, \mathtt{r}', \mathtt{w}', \mathsf{a}'. \; [\mathsf{Inv}^{eq}(\mathtt{r}, \mathtt{w}, \mathsf{a}) \wedge P^{eq}_{\mathtt{Swap}}(\mathtt{r}, \mathtt{w}, \mathsf{a}, \mathtt{r}', \mathtt{w}', \mathsf{a}') \Rightarrow \mathsf{Inv}^{eq}(\mathtt{r}', \mathtt{w}', \mathsf{a}')] \tag{23}$$

of (pure) first-order theory (with equality), where $P^{eq}_{\mathtt{Swap}}$ *is as specified above and* Inv^{eq} *is the translation of* Inv*. If we reason by refutation, Theorem 1 allows us to say that the satisfiability of the negation of (23) modulo* $\mathcal{BA}^e_s$ *is equivalent to the satisfiability of the negation of (3) modulo SSET. It is easy to see that in the negation of (23), the variables* $\mathtt{r}$*,* $\mathtt{w}$*,* a*,* $\mathtt{r}'$*,* $\mathtt{w}'$ *and* a' *become existentially quantified while the variable* $\mathtt{pr}$ *in* $P^{eq}_{\mathtt{Swap}}$ *will still be existentially quantified. As a consequence, all the variables can be replaced by Skolem constants so that the negation of (23) can be considered as a ground formula (see Section 4 for a more systematic handling of quantifiers).*

Theorem 1 can be extended to the restricted class of quantified formulae considered here by using the following observations. First, the function T above acts as the identity on ground terms of sort ELEM. Second, a quantified formula is unsatisfiable iff there exists a ground instance which is unsatisfiable by Herbrand Theorem (see, e.g. [7]). Let G be the function that given an unsatisfiable quantified formula returns its (smallest) unsatisfiable ground instance. Then, it is not difficult to show that ψ is logically equivalent to $\mathsf{G}(\psi')$ modulo $\mathcal{A}^e_s \cup \{\alpha, \alpha'\}$, where $(\psi, \alpha) = \mathsf{F}(\mathsf{G}(\phi))$ and $(\psi', \alpha') = \mathsf{F}^{\mathsf{Q}}(\phi)$.

3.5 Relationships with WS1S

The reader with some knowledge of WS1S [6] may wonder whether *SSET* can be translated to such a logic so that proof obligations can be discharged by existing tools such as, for example, Mona [13]. The answer to this question is indeed positive. However, we point out that the proof obligations generated in our applications are typically very big since the original AM is large. Now, Mona suffers from memory consumptions problem as all tools based on automata do. For this reason, in practice, Mona does not seem suited to handle some of the largest proof obligations arising in our applications (see Section 4.4 for more details on this issue). Furthermore, as already noted above, our approach can be extended to handle more expressive set theories. This is not the case for the translation whose target is the MONA tool. Since WS1S logic can only handle primitive elements (integers) and sets of primitive elements, it is not straightforward to encode in such logic a set of sets of primitive elements. Our theory extends to sets of sets of primitive elements, and so on, simply by extending the theory of arrays $\mathcal{A}^e_s$ accordingly.

4 Discharging Proof Obligations

We are left with the problem of checking that formulae of first-order logic with equality, such as (23), are logical consequences of the equational theory $\mathcal{BA}_s^e$. To do this, we reason by refutation and prove that the negation of the formula is unsatisfiable modulo $\mathcal{BA}_s^e$.

Example 3 *(*The Process Scheduler—continued*) To discharge proof obligation (23), we prove that*

$$\begin{aligned}\exists \mathtt{r}, \mathtt{w}, \mathtt{a}, \mathtt{r}', \mathtt{w}', \mathtt{a}'.\ [(&\mathsf{Inv}^{eq}(\mathtt{r}, \mathtt{w}, \mathtt{a}) \\ &\wedge P^{eq}_{\mathtt{Swap}}(\mathtt{r}, \mathtt{w}, \mathtt{a}, \mathtt{r}', \mathtt{w}', \mathtt{a}') \\ &\wedge \neg \mathsf{Inv}^{eq}(\mathtt{r}', \mathtt{w}', \mathtt{a}')]\end{aligned} \qquad (24)$$

is unsatisfiable modulo $\mathcal{BA}_s^e$.

Our refutation-based theorem proving technique consists of three phases. We describe them in detail in the following subsections.

4.1 Eliminating Quantifiers

We show how to reduce the satisfiability of a first-order formula ϕ (possibly containing quantifiers) modulo $\mathcal{BA}_s^e$ to the satisfiability of a ground formula ϕ_g modulo a theory $\mathcal{EBA}_s^e$ s.t. ϕ is satisfiable modulo $\mathcal{BA}_s^e$ iff ϕ_g is satisfiable modulo $\mathcal{EBA}_s^e$ and $\mathcal{BA}_s^e \subseteq \mathcal{EBA}_s^e$. The key idea is to transform ϕ into a ground formula ϕ_g by replacing the quantified sub-formulae of ϕ with fresh propositional letters and to add their definitions Δ to the axioms $Ax(\mathcal{BA}_s^e)$ of $\mathcal{BA}_s^e$ in such a way that $Ax(\mathcal{BA}_s^e) \wedge \phi$ is satisfiable iff $Ax(\mathcal{BA}_s^e) \wedge \Delta \wedge \phi_g$ is.

Some preliminary definitions (borrowed from [19]) are required. A *position* is a word over the natural numbers. The set $pos(\phi)$ of positions of a formula ϕ is defined as follows: the empty word ϵ is in $pos(\phi)$, $i.p$ is in $pos(\phi)$ if ϕ is of the form $\phi_1 \circ \cdots \circ \phi_n$, $1 \leq i \leq n$, p is in $pos(\phi_i)$, and $\circ$ is a Boolean connective or a quantifier; nothing else is in $pos(\phi)$. We define $\phi|_\epsilon$ to be ϕ and $\phi|_{i.p}$ to be $\phi_i|_p$ when ϕ is of the form $\phi_1 \circ \cdots \circ \phi_n$. We write $\phi[\psi]_p$ when $\phi|_p$ is ψ and $\phi[x/c]$ to denote the formula obtained by replacing all the occurrences of x with c in ϕ. The *polarity* $pol(\phi, \pi)$ of the formula $\phi|\pi$ occurring at position π in a formula ϕ is defined as follows: $pol(\phi, \epsilon) := +1$, $pol(\phi, \pi.i) := pol(\phi, \pi)$ if $\phi|_\pi$ is a conjunction, disjunction, formula whose top-most symbol is a quantifier, an implication with $i = 2$, or an `if then else` with $i = 2$ or $i = 3$, $pol(\phi, \pi.i) := -pol(\phi, \pi)$ if $\phi|_\pi$ is a formula whose top-most symbol is the negation or the implication with $i = 1$, and finally, $pol(\phi, \pi.i) := 0$ if $\phi|_\pi$ is a formula whose top-most symbol is the biconditional or the `if then else` with $i = 1$. We say that the variable x of $Qx.\psi$ is *essentially existentially quantified* in the formula $\phi[Qx.\psi]_\pi$ either when Q is $\forall$ and $pol(\phi, \pi) = -1$ or when Q is $\exists$ and $pol(\phi, \pi) = +1$. Now, we are in the position to describe the details of our algorithm to handle quantifiers.

First of all, a pre-processing step replaces all essentially existentially quantified variables a with fresh Skolem constants $\overline{a}$ and their quantifiers are removed. This can be mechanised by invoking the function $\mathsf{dropExistential}$ which is defined in Figure 2 together with the auxiliary function de whose second parameter is the polarity of the sub-formula being considered (in the Figure, for the sake of conciseness, the mixfix Boolean connective `if then else` has been written as the prefix ternary operator ite). It is easy to check that ϕ is satisfiable iff $\mathsf{dropExistential}(\phi)$ is.

Then, since we want to preserve the propositional structure of the formula as much as possible (so that it can be exploited by haRVey in the following phase), we move quantifiers as far inwards as possible. To this end, we use all the rules to transform a formula into

$$
\begin{aligned}
\mathsf{dropExistential}(\phi) &:= \mathsf{de}(\phi, +1)\\
\mathsf{de}(\phi, p) &:= \phi \text{ if } \phi \text{ is atomic,}\\
\mathsf{de}(\neg\phi, p) &:= \neg\mathsf{de}(\phi, -p)\\
\mathsf{de}(\phi \wedge \psi, p) &:= \mathsf{de}(\phi, p) \wedge \mathsf{de}(\psi, p)\\
\mathsf{de}(\phi \vee \psi, p) &:= \mathsf{de}(\phi, p) \vee \mathsf{de}(\psi, p)\\
\mathsf{de}(\mathsf{ite}(\phi, \psi, \xi), p) &:= \mathsf{ite}(\phi, \mathsf{de}(\psi, p), \mathsf{de}(\xi, p))\\
\mathsf{de}(\phi \Rightarrow \psi, p) &:= \mathsf{de}(\phi, -p) \Rightarrow \mathsf{de}(\psi, p)\\
\mathsf{de}(\phi \Leftrightarrow \psi, p) &:= \phi \Leftrightarrow \psi\\
\mathsf{de}(\forall x.\phi, +1) &:= \forall x.\phi\\
\mathsf{de}(\exists x.\phi, -1) &:= \exists x.\phi\\
\mathsf{de}(\forall x.\phi, -1) &:= \mathsf{de}(\phi[x/c], +1)\\
\mathsf{de}(\exists x.\phi, +1) &:= \mathsf{de}(\phi[x/c], -1)
\end{aligned}
$$

Figure 2: Definition of $\mathsf{dropExistential}$ and de.

prenex form[9] (see again [12] for details) but in the opposite direction. For example, the formula $\exists x.(\phi \Rightarrow \psi)$ is transformed to $(\forall x.\phi) \Rightarrow \psi$ if x does not occur in ψ. This operation, known as *miniscoping*, can be mechanised as shown in Figure 3. We assume that only formulae obtained as the result of the invocation of $\mathsf{dropExistential}$ are fed to mini. The function is defined on the structure of first-order formulae. When it encounters an atomic formula it simply returns it whereas when a quantified sub-formula is met, the auxiliary function m_q is invoked. In the definition of m_q, the following notation is used: Q abbreviates either $\forall$ or $\exists$, $-Q$ abbreviates $\forall$ ($\exists$) when Q is $\exists$ ($\forall$), $\mathcal{V}(\phi)$ denotes the set of free variables in ϕ and x' is a fresh variable. The first and the second arguments of m_q are the quantifier and the quantified variable whose scope is reduced by the function while the third argument is the already processed sub-formula. Again, it

[9] A formula is in prenex form if it has the structure $Q_1x_1...Q_nx_n.\phi$, where Q_i is either $\forall$ or $\exists$, x_i is a variable ($i = 1, ..., n$), and ϕ is a quantifier-free formula whose free variables are $x_1, ..., x_n$.

$$
\begin{aligned}
\mathsf{mini}(\phi \circ \psi) &:= \mathsf{mini}(\phi) \circ \mathsf{mini}(\psi) \text{ where } \circ \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\}\\
\mathsf{mini}(\neg\phi) &:= \neg\mathsf{mini}(\phi)\\
\mathsf{mini}(\mathsf{ite}(\phi, \psi, \xi)) &:= \mathsf{ite}(\mathsf{mini}(\phi), \mathsf{mini}(\psi), \mathsf{mini}(\xi))\\
\mathsf{mini}(\phi) &:= \phi \text{ if } \phi \text{ is atomic}\\
\mathsf{mini}(Q\ x_1, \cdots, x_n.\phi) &:= \mathsf{m}_q(Q, x_1, \ldots \mathsf{m}_q(Q, x_n, \mathsf{mini}(\phi)) \ldots)\\
\mathsf{m}_q(Q, x, \phi) &:= \phi \text{ if } x \notin \mathcal{V}(\phi)\\
\mathsf{m}_q(Q, x, \neg\phi) &:= \neg\mathsf{m}_q(-Q, x, \phi)\\
\mathsf{m}_q(Q, x, \phi \diamond \psi) &:= \mathsf{m}_q(Q, x, \phi) \diamond \psi \text{ if } x \notin \mathcal{V}(\psi),\ \diamond \in \{\wedge, \vee\}\\
\mathsf{m}_q(Q, x, \phi \diamond \psi) &:= \phi \diamond \mathsf{m}_q(Q, x, \psi) \text{ if } x \notin \mathcal{V}(\phi),\ \diamond \in \{\wedge, \vee\}\\
\mathsf{m}_q(\forall, x, \phi \wedge \psi) &:= \mathsf{m}_q(\forall, x, \phi) \wedge \mathsf{m}_q(\forall, x', \psi[x/x']) \text{ otherwise}\\
\mathsf{m}_q(\exists, x, \phi \vee \psi) &:= \mathsf{m}_q(\exists, x, \phi) \vee \mathsf{m}_q(\exists, x', \psi[x/x']) \text{ otherwise}\\
\mathsf{m}_q(Q, x, \phi \Rightarrow \psi) &:= \phi \Rightarrow \mathsf{m}_q(Q, x, \psi) \text{ if } x \notin \mathcal{V}(\phi)\\
\mathsf{m}_q(Q, x, \phi \Rightarrow \psi) &:= \mathsf{m}_q(-Q, x, \phi) \Rightarrow \psi \text{ if } x \notin \mathcal{V}(\psi)\\
\mathsf{m}_q(\exists, x, \phi \Rightarrow \psi) &:= \mathsf{m}_q(\forall, x, \phi) \Rightarrow \mathsf{m}_q(\exists, x', \psi[x/x'])\\
\mathsf{m}_q(Q, x, \mathsf{ite}(\phi, \psi, \xi)) &:= \mathsf{ite}(\phi, \psi, \mathsf{m}_q(Q, x, \xi)) \text{ if } x \notin \mathcal{V}(\phi) \cup \mathcal{V}(\psi)\\
\mathsf{m}_q(Q, x, \mathsf{ite}(\phi, \psi, \xi)) &:= \mathsf{ite}(\phi, \mathsf{m}_q(Q, x, \psi), \xi) \text{ if } x \notin \mathcal{V}(\phi) \cup \mathcal{V}(\xi)\\
\mathsf{m}_q(Q, x, \phi) &:= Q\ x.\phi \text{ otherwise}
\end{aligned}
$$

Figure 3: Definition of mini and m_q.

is a routine exercise to check that ϕ is satisfiable iff $\mathsf{mini}(\phi)$ is.

As the third and last step, we replace each outermost quantified sub-formula ψ with a fresh propositional letter q and we add $q \Rightarrow \psi$ (resp. $\psi \Rightarrow q$, $q \Leftrightarrow \psi$) to $Ax(\mathcal{BA}_s^e)$ if $\phi|_\pi = \psi$ and $pol(\phi, \pi) = +1$ (resp. $pol(\phi, \pi) = -1$, $pol(\phi, \pi) = 0$). Notice that if there is a quantified sub-formula ψ' occurring in ψ, we do not recursively apply the above process to ψ' but we stop at ψ. This is a heuristic decision which seems to give good results in practice. The mechanization of this last phase is given in Figure 4. We assume that only formulae obtained from the applications of $\mathsf{dropExistential}$ first followed by mini are passed to $\mathsf{renameFormula}$. The first argument of the auxiliary function rf is the formula being considered and the second is its polarity. The function

$$
\begin{array}{rcl}
\mathsf{renameFormula}(\phi) & := & \mathsf{rf}(\phi, +1) \\
\mathsf{rf}(\phi, p) & := & (\phi, \emptyset) \text{ if } \phi \text{ is atomic} \\
\mathsf{rf}(\neg\phi, p) & := & (\neg\psi, \mathcal{A}) \\
 & & \text{where } (\psi, \mathcal{A}) = \mathsf{rf}(\phi, -p) \\
\mathsf{rf}(\phi_1 \diamond \phi_2, p) & := & (\psi_1 \diamond \psi_2, \mathcal{A}_1 \cup \mathcal{A}_2) \\
 & & \text{where } (\psi_i, \mathcal{A}_i) = \mathsf{rf}(\phi_i, p) \\
\mathsf{rf}(\mathsf{ite}(\phi_1, \phi_2, \phi_3), p) & := & (\mathsf{ite}(\psi_1, \psi_2, \psi_3), \mathcal{A}_1 \cup \mathcal{A}_2 \cup \mathcal{A}_3) \\
 & & \text{where } (\psi_1, \mathcal{A}_1) = \mathsf{rf}(\phi_1, 0), \\
 & & (\psi_2, \mathcal{A}_2) = \mathsf{rf}(\phi_2, p) \\
 & & \text{and } (\psi_3, \mathcal{A}_3) = \mathsf{rf}(\phi_3, p) \\
\mathsf{rf}(\phi_1 \Rightarrow \phi_2, p) & := & (\psi_1 \Rightarrow \psi_2, \mathcal{A}_1 \cup \mathcal{A}_2) \\
 & & \text{where } (\psi_1, \mathcal{A}_1) = \mathsf{rf}(\phi_1, -p) \\
 & & \text{and } (\psi_2, \mathcal{A}_2) = \mathsf{rf}(\phi_2, p) \\
\mathsf{rf}(\phi_1 \Leftrightarrow \phi_2, p) & := & (\psi_1 \Leftrightarrow \psi_2, \mathcal{A}_1 \cup \mathcal{A}_2) \\
 & & \text{where } (\psi_i, \mathcal{A}_i) = \mathsf{rf}(\phi_i, 0) \\
\mathsf{rf}(Qx.\phi, 0) & := & (p, \{p \Leftrightarrow Qx.\phi\}) \\
\mathsf{rf}(Qx.\phi, +1) & := & (p, \{p \Rightarrow (Qx.\phi)\}) \\
\mathsf{rf}(Qx.\phi, -1) & := & (p, \{(Qx.\phi) \Rightarrow p\})
\end{array}
$$

Figure 4: Definition of renameFormula and rf.

returns a pair whose first argument is the result of replacing quantified sub-formulae with fresh propositional letters and the second is the set of implications between quantified sub-formulae and propositional letters to be added to the background theory. We can show that ϕ is satisfiable iff $\phi' \wedge \Phi$ is, where $(\phi', \Phi) = \mathsf{renameFormula}(\phi)$. The proof is by induction on the structure of the formula ϕ. To illustrate, we consider the case where ϕ is of the form $\xi \vee \exists x.\psi$ s.t. $\mathsf{renameFormula}(\phi) = (\xi \vee q, \{q \Rightarrow \exists x.\psi\})$, where q is a fresh propositional letter. Now, any model of $(\xi \vee q) \wedge (q \Rightarrow \exists x.\psi)$ is also a model of $\xi \vee \exists x.\psi$ since this last is a logical consequence of $\xi \vee q$ and $q \Rightarrow \exists x.\psi$. Conversely, a model $\mathcal{M}'$ for $(\xi \vee q) \wedge (q \Rightarrow \exists x.\psi)$ can be obtained from a model $\mathcal{M}$ for $\xi \vee \exists x.\psi$ by defining q s.t. $\mathcal{M}'$ satisfies $q \Leftrightarrow \exists x.\psi$. The remaining cases are similar and therefore omitted.

Let $(\phi_g, \Phi_g) = \mathsf{renameFormula}(\mathsf{mini}(\mathsf{dropExistential}(\phi)))$ and $\mathcal{EBA}^e_s$ be the theory axiomatized by $Ax(\mathcal{BA}^e_s) \cup \Phi_g$.

Theorem 2 *ϕ is satisfiable modulo $\mathcal{BA}^e_s$ iff (the ground formula) ϕ_g is satisfiable modulo $\mathcal{EBA}^e_s$.*

The proof of this theorem is an immediate consequence of the fact that each function preserves satisfiability as argued above.

Example 4 *(*The Process Scheduler—continued*) Let us consider formula (24). First of all, we replace the outermost existentially quantified variables with fresh constants obtaining the formula:*

$$\mathsf{Inv}^{eq}(\overline{\mathtt{r}}, \overline{\mathtt{w}}, \overline{\mathtt{a}}) \wedge P^{eq}_{\mathtt{Swap}}(\overline{\mathtt{r}}, \overline{\mathtt{w}}, \overline{\mathtt{a}}, \overline{\mathtt{r}}', \overline{\mathtt{w}}', \overline{\mathtt{a}}') \wedge \neg\mathsf{Inv}^{eq}(\overline{\mathtt{r}}', \overline{\mathtt{w}}', \overline{\mathtt{a}}').$$

which is satisfiable iff (24) is (by basic properties of first-order logic [12]).[10] *The sub-formula $P^{eq}_{\mathtt{Swap}}(\overline{\mathtt{r}}, \overline{\mathtt{w}}, \overline{\mathtt{a}}, \overline{\mathtt{r}}', \overline{\mathtt{w}}', \overline{\mathtt{a}}')$ in the formula above is left unmodified since the only existential quantifier cannot be further moved inward and we replace the existentially quantified sub-formula by the fresh propositional letter q. Thus, we obtain the following equisatisfiable formula*

$$
\begin{array}{l}
\mathsf{if}\ \ \overline{\mathsf{a}} \neq \mathsf{mty}\ \ \mathsf{then} \\
\quad \overline{\mathtt{w}}' = \mathtt{w\_U\_a} \wedge \\
\quad (\mathsf{if}\ \ \overline{\mathtt{r}} \neq \mathsf{mty}\ \ \mathsf{then}\ \ q\ \ \mathsf{else}\ \ \overline{\mathsf{a}}' = \mathsf{mty} \wedge \overline{\mathtt{r}}' = \overline{\mathtt{r}}) \\
\mathsf{else} \\
\quad \overline{\mathsf{a}}' = \overline{\mathsf{a}} \wedge \overline{\mathtt{r}}' = \overline{\mathtt{r}} \wedge \overline{\mathtt{w}}' = \overline{\mathtt{w}}
\end{array}
$$

and the formula $q \Rightarrow \exists \mathtt{pr}.(\mathsf{rd}(\overline{\mathtt{r}}, \mathtt{pr}) = \mathsf{tt} \wedge \overline{\mathsf{a}}' = \mathsf{wr}(\mathsf{mty}, \mathtt{pr}, \mathsf{tt}) \wedge \overline{\mathtt{r}}' = \mathsf{wr}(\overline{\mathtt{r}}, \mathtt{pr}, \mathsf{ff}))$ is added to $Ax(\mathcal{BA}^e_s)$.

[10]If the formula contains free variables we take its existential closure.

4.2 Checking Satisfiability

We are left with the problem of checking the unsatisfiability of the ground formula ϕ_g modulo the first-order theory $\mathcal{EBA}_s^e$. We solve this problem by invoking haRVey[11], a tool based on the flexible and efficient combination of BDDs and superposition theorem proving (see [10] for details). The idea is to abstract ground atoms to propositional letters and then let BDDs represent the Boolean structure of (an abstraction of) ϕ_g. Since it is easy to extract the Disjunctive Normal Form (DNF) of ϕ_g from its BDD representation, we check the satisfiability of each disjunct in the DNF modulo $\mathcal{EBA}_s^e$ by invoking a superposition theorem prover. In practice, a refinement of this schema which greatly improves performances (based on the capability of generating suitable lemmas to simplify the BDD) is implemented in the system. In order to build procedures which check for the satisfiability modulo a first-order theory, we adopt the superposition-based approach of [3]. This permits the flexible implementation of many decision (and semi-decision) procedures by simply feeding a superposition theorem prover with the axioms of the theory and the literals to be proved satisfiable. It is also an efficient alternative to specialized decision procedures as shown in [2, 10].

The algorithm underlying haRVey is shown in Figure 5. Let *Atm* be the set of distinct ground atoms in ϕ_g and P_{Atm} be a set of propositional letters s.t. the cardinalities of *Atm* and P_{Atm} are equal and $Atm \cap P_{Atm} = \emptyset$. Let *atm2pl* be a bijective function from *Atm* to P_{Atm}. We define the mapping *fol2prop* from the ground first-order formula ϕ_g to a propositional formula as the homeomorphic extension of *atm2pl* to ϕ_g. Then, *abs* takes the propositional formula returned by *fol2prop* and builds its BDD representation ϕ^a. The function *pickBranch* selects a branch β^a of the BDD ϕ^a going from the root to the node labelled by *true*; β^a is a conjunction of propositional literals. We assume that *prop2fol* is the inverse of *fol2prop* so that its result is a conjunction of ground first-order literals. Furthermore, we require that $checkSatBranch(Ax(\mathcal{EBA}_s^e), \beta) = (\perp, \pi)$ iff π is a sub-formula of β and it is unsatisfiable modulo $\mathcal{EBA}_s^e$; thereby *checkSatBranch* is a decision procedure for the problem of checking whether a conjunction of ground literals is satisfiable modulo $\mathcal{EBA}_s^e$ with the capability of returning small subsets of the input literals which are unsatisfiable. Such a set of literals can then be used to simplify the BDD, as shown by the last statement of the loop in Figure 5. We consider the branches (leading to the leaf labelled with true) in the BDD until one is shown to be satisfiable modulo $\mathcal{EBA}_s^e$ (in which case, we return that ϕ_g is satisfiable) or the BDD has been reduced to $\perp$ (i.e. false) by interleaving unsatisfiability checking and BDD simplification steps.

```
function haRVeyCheckSat ($\mathcal{EBA}_s^e$: first-order theory,
                         $\phi_g$: ground formula)
  $\phi^a \leftarrow abs(fol2prop(\phi_g))$
  while $\phi^a \neq \perp$ do
      $\beta^a \leftarrow pickBranch(\phi^a)$
      $(\rho, \pi) \leftarrow checkSatBranch(Ax(\mathcal{EBA}_s^e), prop2fol(\beta^a))$
      if $\rho \neq \perp$ then return yes
      $\phi^a \leftarrow \phi^a \wedge \neg fol2prop(\pi)$
  return no
end
```

Figure 5: The Algorithm of haRVey.

[11] http://www.loria.fr/equipes/cassis/softwares/haRVey

4.3 Providing the User with Counter-Examples

If a branch β has been found satisfiable by haRVey, then we can use it as the starting point to build a model for the formula ϕ_g under consideration, i.e. a counter-example for the formula to be proved valid (which is the negation of ϕ_g). Notice that this is an advantage w.r.t. building a model for the whole formula ϕ_g since the branch is usually much smaller.

In preliminary investigations, we have tried to build a (finite) model of β by using state-of-the-art model finders (such as MACE[12] and SEM[13]) without success. As a matter of fact, the theory $\mathcal{EBA}_s^e$ generates a search space which is too large to be treated in a reasonable amount of time. In order to overcome these difficulties, we have investigated the possibility to use the CLPS tool [5], a constraint solver for the theory of Hereditarily Finite Sets with Atoms which includes *SSET*. This tool has already been successfully used on industrial verification problems. Since the input language of CLPS is based on set theoretic constructs, we need to translate the branch β back to a conjunction η of literals in *SSET*. Once the universal set of the AM (cf. the set `PID` in Figure 1) is manually instantiated to a certain finite set, CLPS can find a solution (if any) to η which (possibly) constitutes a transition of (an instance of) the AM leading to a state which violates the invariant. However, notice that we are not guaranteed that such a state is reachable by executing the AM starting in the initial state (specified by the clause `INITIALISATION` in Figure 1). To establish whether the state is reachable or not, two solutions are possible. First, one can use an animation tool (e.g. the one integrated in [14]) in order to discover if there exists an execution of the AM going from the initial state to the one found by CLPS. Second, use a model checker (such as SMV) to check whether the state found by CLPS is reachable from the initial state. In this second case, we should translate the AM to the input language of the model checker. This should be possible because we are considering a finite instance of the original AM since we have instantiated to a finite set the universal set in the AM in order to invoke CLPS. Once the reachability of the state found by CLPS is established, the counter-example can be used as the basis to correct the initial AM, either by changing the system or by strengthening the invariant.

```
Ready =
  ANY pw WHERE pw ∈ waiting
  THEN
    /* waiting := waiting - {pw} || */
    IF (active ≠ ∅) THEN
      ready := ready ∪ {pw}
    ELSE
      active := {pw}
    END
  END
```

Figure 6: An Erroneous Specification of the Operation *Ready*.

Example 5 (The Process Scheduler—continued) *To illustrate, we consider the (erroneous) specification of the Ready operation given in Figure 6, which must be considered as part of the AM in Figure 1. In this operation, a process* `pw`*, that has been waiting, is somehow unblocked and can go to the ready state or to the active state, if the system is idle. The bug consists of leaving* `pw` *in* `waiting`*. As a result, the invariant is violated since the intersection of* `waiting` *and* `active` *at the next state will contain* `pw` *and will no more be*

[12] `http://www-unix.mcs.anl.gov/AR/mace2/`
[13] `http://www.cs.uiowa.edu/~hzhang/sem.html`

empty. (Notice that to correct the problem it is sufficient to uncomment the line delimited by `/*` *and* `*/` *in Figure 6.) We want to detect the anomalous situation by using the approach described above. First, we generate the verification condition (along the lines of Section 2). Then, we translate and manipulate the resulting formula so that haRVey can process it (cf. Sections 3 and 4). The system finds that the proof obligation is not valid and returns the following set of literals (intended conjunctively):* $\{\mathtt{q}_0, \mathtt{q}_1, \mathtt{q}_2, \mathtt{q}_3, \mathtt{active} = \mathtt{mty}, \mathtt{waiting}' = \mathtt{waiting}, \mathtt{ready}' = \mathtt{ready}, \mathtt{active}' = \mathtt{i}_{25}, \mathtt{rd}(\mathtt{waiting}, \mathtt{pw}) = \mathtt{tt}, \mathtt{i}_{17} = \mathtt{mty}, \mathtt{i}_{14} = \mathtt{mty}, \mathtt{i}_{11} = \mathtt{mty}, \mathtt{i}_{35} \neq \mathtt{mty}\}$.[14] *As the reader can see, it is quite difficult to understand the problem in the specification by looking at this set of literals, even for this simple example. As a matter of fact, it is rather obscure to see the meaning of the propositional letters* $\mathtt{q}_0, ..., \mathtt{q}_3$ *and of the constants* $\mathtt{i}_{25}, ..., \mathtt{i}_{35}$ *which have been introduced to eliminate set-theoretic constructs and quantifiers as specified in Sections 3 and 4.1. However, it is easy to maintain a table (during the translation) which associates the original set theoretic literals with the literals passed to haRVey. The table for the formula under consideration is shown in Table 1. The boxed literal corresponds to the violation of the invariant, namely the intersection of* `waiting` *and* `active` *at the next state will be non-empty (contrary to what is stated in the clause* `INVARIANT` *of Figure 1). Now, we can invoke the CLPS solver on the conjunctions of the literals in the right column of Table 1 after having instantiated the universal set* `PID` *to be the singleton set* $\{\mathtt{p1}\}$. *The solver returns the following solution for such a set of constraints:*

$$\begin{cases} \mathtt{waiting} = \mathtt{waiting}' = \mathtt{active}' = \{\mathtt{p1}\} \\ \mathtt{active} = \mathtt{ready} = \mathtt{ready}' = \emptyset. \end{cases} \tag{25}$$

Then, the user is free to run an animation tool so to check that the state identified by (25) is reachable from the initial state given in Figure 1. Afterwards, he/she can make modification to the operation (sometimes, it is also required to strengthen the invariant to make it inductive) so that the generated proof obligation will be found valid. After some experiments with the animation tool, it is easy to see that all we need to do is to add the commented line of Figure 6 to correct the bug.

haRVey Literal	Set-theoretic Literal
$\mathtt{q}_0$	$\mathtt{waiting} \subseteq \mathtt{PID}$
$\mathtt{q}_1$	$\mathtt{ready} \subseteq \mathtt{PID}$
$\mathtt{q}_2$	$\mathtt{active} \subseteq \mathtt{PID}$
$\mathtt{q}_3$	$\lvert\mathtt{active}'\rvert = 1$
$\mathtt{active} = \mathtt{mty}$	$\mathtt{active} = \emptyset$
$\mathtt{waiting}' = \mathtt{waiting}$	$\mathtt{waiting}' = \mathtt{waiting}$
$\mathtt{ready}' = \mathtt{ready}$	$\mathtt{ready}' = \mathtt{ready}$
$\mathtt{active}' = \mathtt{i}_{25}$	$\mathtt{active}' = \{\mathtt{pw}\}$
$\mathtt{rd}(\mathtt{waiting}, \mathtt{pw}) = \mathtt{tt}$	$\mathtt{pw} \in \mathtt{waiting}$
$\mathtt{i}_{17} = \mathtt{mty}$	$\mathtt{active} \cap \mathtt{waiting} = \emptyset$
$\mathtt{i}_{14} = \mathtt{mty}$	$\mathtt{ready} \cap \mathtt{waiting} = \emptyset$
$\mathtt{i}_{11} = \mathtt{mty}$	$\mathtt{ready} \cap \mathtt{active} = \emptyset$
$\mathtt{i}_{35} \neq \mathtt{mty}$	$\boxed{\mathtt{active}' \cap \mathtt{waiting}' \neq \emptyset}$

Table 1: Associations between haRVey Literals and Atoms of Set-Theory.

4.4 Prototype Implementation

The architecture of our system is depicted in Figure 7. The box with round corners contains the functionalities we have implemented so far and the boxes outside

[14]The output of haRVey has been slightly edited in order to improve the readability.

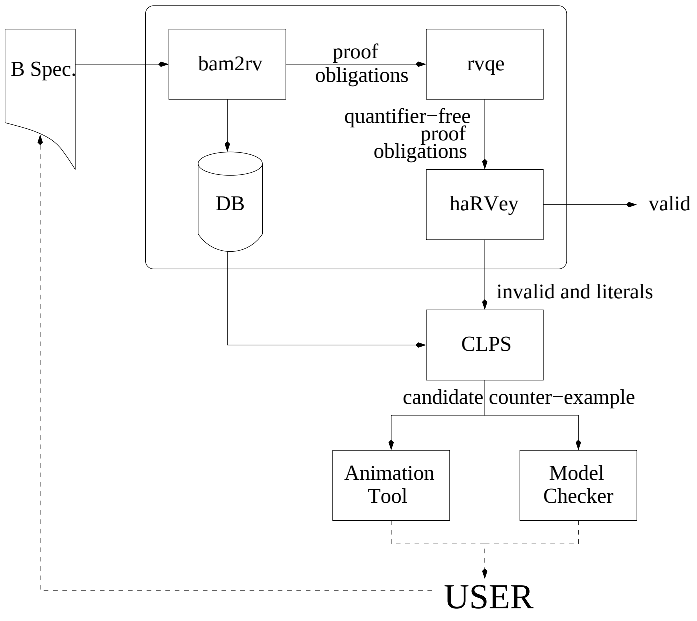

Figure 7: The Architecture of our System.

are the tools which are already available. `bam2rv`[15] takes a B abstract machine and returns a set of proof obligations in pure equational first-order logic along the lines of Section 3. `rvqe`[16] is a pre-processor to eliminate the quantifiers from the proof obligations along the lines of Section 4.1. The second and the last authors are also developing haRVey [10] as a flexible and efficient reasoning module to be used in larger verification tools, as it is the case for the application described here. `bam2rv` and `rvqe` are implemented in Java while haRVey is developed in C. The various tools exchange information by using the ATerms data structure.[17] Let us briefly analyse the flow of the data in the architecture. The specification of a B machine annotated with an invariant is given to `bam2rv`. This module translates the various B operations into before-after predicates and constructs the proof obligations entailing that the specified invariant is an inductive invariant of the system by eliminating set-theoretic operators. These proof obligations are formulae of first-order logic with equality possibly containing quantifiers. Notice also that the association between literals containing set-theoretic constructs and their translation to formulae of pure equational first-order logic are stored in DB. Then, the negation of the proof obligations are sent to `rvqe` which eliminates the quantifiers occurrences and builds a rich background theory so as to take into account the quantified subformulae. These ground formulae and theory are sent to `haRVey` which is capable of proving their satisfiability or unsatisfiability. If these formulae are all unsatisfiable, then we are entitled to conclude that the original proof obligations are valid and the specified invariant is an inductive invariant of the system. If a formula is satisfiable, then a conjunction of literals in the formula is returned by haRVey from which a counter-example can be built. To do this, we retrieve from DB the set-theoretic literals associated with the literals returned by `haRVey` and we invoke the CLPS solver on the resulting set of literals. This will return a candidate solution representing a state of the AM which falsifies the invariant. Then by means of an animation tool or a model checker, we can check whether the state is reachable from the initial state or not. If so, the error trace is shown to the user which can modify the specification in order to correct the problem. If the state is not reachable, then we ask the CLPS solver to return a new solution and we check whether the new state is reachable and so on. The prototype tool is indeed capable of discharging all the proof obligations of the process scheduler

[15]`http://lifc.univ-fcomte.fr/~giorgett/Rech/Software/bam2rv/`

[16]`http://lifc.univ-fcomte.fr/~couchot/rvqe`

[17]`http://www.cwi.nl/htbin/sen1/twiki/bin/view/SEN1/ATermLibrary`

discussed here and to detect the invalidity of those generated from some buggy versions.

4.5 Experimental Results

Table 2 presents a comparison of an implementation of our approach with AtelierB.[18] The first column ("Model") lists the identifier of the considered B AMs. The columns haRVey, bam2rv, and rvqe give the partial time spent in each module of the system while the column "Total" lists the total time to prove the invariant of the machine (which is the sum of the times in the previous three columns). The column AtelierB gives the time spent by this commercial tool on the various machines.

Four problems have been considered: "Pidset" is the running example used in this paper, "Robot" is a device to move blocks around, "T=1" is a communication protocol between a smart card and its reader, and "SCard" specifies a smart card terminal developed by an industrial partner. The first three problems are small (at most one A4-page) while the last consists of about six A4-pages without comments. We stress here that "SCard" is the perfect example of the kind of specifications our technique has been designed for, i.e. large models with a rich control path (giving rise to a large and complex Boolean structure of the resulting proof obligation obligations) performing operations on relatively simple data structures (which require only selected fragments of set theory, such as *SSET*, to be specified).

Model	bam2rv	rvqe	**haRVey**	Total	AtelierB
Pidset	3.5	12.3	5.8	21.6	6.0
Robot	3.3	15.0	6.7	25.0	2.1
T=1	3.6	22.9	43.1	69.8	1.0
SCard	7.0	26.2	192.0	225.2	$\surd$

Table 2: Experimental results.

Experience. AtelierB (we have used version 3.6_BOM.1) is one order of magnitude faster than our tool on the three small problems but it aborts (cf. $\surd$) after 45 minutes on the last. The reasons for these results are mainly two. First, the tools bam2rv and rvqe are still prototypes which have not been coded with efficiency in mind. We believe that their execution time can be reduced of one order of magnitude with some care. This would yield a substantial speed-up for the first three examples where the execution times of bam2rv and rvqe represent from 50% to 70% of the total time. Second, our system scales up more smoothly to the industrial specification "SCard" than AtelierB because of the integration between Boolean reasoning (the BDD library) and first-order reasoning (the E prover) featured by haRVey. In particular, we believe that its capability of pruning the BDD representing the Boolean structure of the proof obligation by using the information received from the prover is the key to the scalability. In fact, AtelierB finds large proof obligations with rich Boolean structures difficult to handle because it tries to break them down to small formulae with a very simple Boolean structure by using a Tableau-like technique (see e.g. [8]), whose efficiency is well-known to be inferior to BDDs. Furthermore, it does not exploit the proof of a sub-formula to prune the search space of the Tableau procedure and this may result in the generation of a dramatic number of formulae to be discharged by the first-order reasoner. This analysis is confirmed by the

[18]All the experiments have been run on a Pentium II 250 Mhz running Linux with 128 Mb of RAM. Timings are expressed in seconds.

fact that for "SCard", AtelierB generates more than $10,000$ formulae (its maximal bound) after about 45 minutes.

We have also built a prototype tool which encodes the proof obligations in the logic of WS1S[19] so that the system Mona [13] can be invoked. On "SCard", Mona runs out of memory after two hours of computation, having built more than 140 automata in memory.

We believe that these preliminary results confirm the viability of our approach.

5 Conclusion and Future Work

We have presented a technique to prove invariants of model-based specifications in a fragment of set theory. Proof obligations containing set theory constructs are translated to first-order logic with equality augmented with (an extension of) the theory of arrays with extensionality. A theorem proving procedure automating the verification of the proof obligations obtained by the translation has also been described. The technique has been implemented and experimental results confirm the viability of our approach.

The lines of future research are essentially fourfold. First, we envisage extending the decidability result for $\mathcal{A}_s^e$ in [3] to the theory $\mathcal{BA}_s^e$ considered in this paper. Second, we plan to handle a larger number of constructs: on the one hand we will integrate operators as Cartesian product and relations which are commonly used in state-based specifications and, on the other hand, we will add some B syntactic sugar. Third, we want to integrate the CLPS solver in our tool so that meaningful counter-examples can be automatically built from failed proof attempts. Finally, we plan to apply our technique to a larger number of case studies.

[19]This is possible as observed in Section 3.5.

Acknowledgement

We thank Frédéric Dadeau for his implementation of bam2rv and many useful discussions on the work reported in the paper. We would also like to thank Dominique Cansell and Stephan Merz for useful comments on a preliminary version of this paper.

References

[1] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.

[2] A. Armando, M.P. Bonacina, S. Ranise, M. Rusinowitch, and A. K. Sehgal. High-Performance Deduction for Verification: A Case Study in the Theory of Arrays. In *Proc. of VERIFY'02 (FLoC'02 Affiliated Wokshop)*, 2002.

[3] A. Armando, S. Ranise, and M. Rusinowitch. A Rewriting Approach to Satisfiability Procedures. *Info. and Comp.*, 183(2):140–164, June 2003.

[4] J.-P. Bodeveix and M. Filali. Type Synthesis in B and the Translation of B to PVS. In *Proc. of ZB 2002*, volume 2272 of *LNCS*, pages 350–369. Springer Verlag, 2002.

[5] F. Bouquet, B. Legeard, and F. Peureux. CLPS-B - A Constraint Solver for B. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems, TACAS2002*, volume 2280 of *LNCS*, pages 188–204. Springer Verlag, 2002.

[6] J. R. Büchi. Weak second order arithmetic and finite automata. *Zeitschr. f. math. Logik und Grundlagen d. Math.*, 6:66–92, 1960.

[7] C.-L. Chang and R. C.-T. Lee. *Symbolic Logic and Mechanical Theorem Proving.* Academic Press, 1973.

[8] M. D'Agostino, D. M. Gabbay, R. Hhnle, and J. Posegga, editors. *Handbook of Tableau Methods.* Kluwer Ac. Publ., Dordrecht, 1999.

[9] J. Dawes. *The VDM-SL Reference Guide.* Pitman, 1991.

[10] D. Déharbe and S. Ranise. Light-weight theorem proving for debugging and verifying units of code. In *International Conference on Software Engineering and Formal Methods (SEFM03).* IEEE Computer Society Press, 2003.

[11] J. Dick and A. Faivre. Automating the Generation and Sequencing of Test Cases from Model-Based Specifications. In *Proc. of FME'93*, volume 670 of *LNCS*, pages 268–284. Springer Verlag, 1993.

[12] H. B. Enderton. *A Mathematical Introduction to Logic.* Ac. Press, Inc., 1972.

[13] J. G. Henriksen, J. L. Jensen, M. E. Jørgensen, N. Klarlund, R. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *Proc. of Tools and Algorithms for the Construction and Analysis of Systems*, volume 1019 of *LNCS*, pages 89–110. Springer Verlag, 1996.

[14] M. Leuschel and M. Butler. The ProB Animator and Model Checker for B. In *Proc. 12th International FME Symposium 2003 (FM2003)*, 2003.

[15] L. Mikhailov and M. Butler. An Approach to Combining B and Alloy. In *Proc. of ZB 2002*, volume 2272 of *LNCS*, pages 140–161. Springer Verlag, 2002.

[16] R. Nieuwenhuis and A. Rubio. Paramodulation-based theorem proving. In A. Robinson and A. Voronkov, editors, *Hand. of Automated Reasoning.* 2001.

[17] N. Shankar. Little Engines of Proof. In *Formal Methods Europe (FME'02)*, volume 2391 of *LNCS*, pages 1–20. Springer-Verlag, 2002.

[18] M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, 2nd edition, 1992.

[19] C. Weidenbach and A. Nonnengart. Handbook of automated reasoning, vol.1, 2001. A. Robinson and A. Voronkov editors. Elsevier Science.